Effective Concurrency Testing for Go via Directional Primitive-constrained Interleaving Exploration

Zongze Jiang[1†], Ming Wen[1†*], Yixin Yang[1†], Chao Peng[3], Ping Yang[3], Hai Jin[2†]
[1]*School of Cyber Science and Engineering, Huazhong University of Science and Technology, China*
[2]*School of Computer Science and Technology, Huazhong University of Science and Technology, China*
[3]*ByteDance, Beijing, China*
Email: {jiangzongze, mwenaa, yangyixin, hjin}@hust.edu.cn, {pengchao.x, yangping.cser}@bytedance.com

***Abstract*—The Go language (Go/Golang) has been attracting increasing attention from the industry over recent years due to its strong concurrency support and ease of deployment. This programming language encourages developers to use channel-based concurrency, which simplifies the development of concurrent programs. Unfortunately, it also introduces new concurrency problems that differ from those caused by the mechanism of shared memory concurrency. However, there are only few works that aim to detect such Go-specific concurrency issues. Even state-of-the-art testing tools will miss critical concurrent bugs that require fine-grained and effective interleaving exploration.**

This paper presents GoPie, a novel testing approach for detecting Go concurrency bugs through primitive-constrained interleaving exploration. GoPie utilizes execution histories to identify new interleavings instead of relying on exhaustive exploration or random scheduling. To evaluate its performance, we applied GoPie to existing benchmarks and large-scale open-source projects. Results show that GoPie can effectively explore concurrent interleavings and detect significantly more bugs in the benchmark. Furthermore, it uncovered 11 unique previously unknown concurrent bugs, and 9 of which have been confirmed.

***Index Terms*—Go, Concurrency Testing, Fuzzing**

I. Introduction

The Go language offers two concurrency mechanisms: Communicating Sequential Processes (CSP) and traditional ones based on shared memory. CSP is a concurrency model in Go that enables the communication between independent processes based on *channels*. A *channel* acts as a pipeline to connect different processes, allowing them to coordinate different tasks and share data. The usage of *channels* makes the complex synchronization operations transparent to users, making it easier to implement concurrent programs than other programming languages without such built-in supports. Consequently, since its open source in 2009, the Go language has consistently grown in popularity. However, recent works [1], [2], [3] have shown that the utilization of channels does not eliminate concurrency bugs. Concurrency bugs are still pervasive due to the insufficient understanding of such new mechanisms, which can lead to resource leaks and global/partial deadlocks, thus eventually leading to severe consequences for end-users. Therefore, effective detection of channel-related concurrency bugs in Go automatically is in urgent demand.

```
1  type statusManager struct {
2      podStatusesLock  sync.Mutex
3      podStatusChannel chan bool
4      // ...
5  }
6
7  func(s *statusManager)Start() {
8      for i:=0; i<2;i++{
9          <-s.podStatusChannel
10         s.podStatusesLock.Lock()
11         // handle the pod status here
12         s.podStatusesLock.Unlock()
13     }
14 }
15
16 func(s *statusManager)SetPodStatus() {
17     s.podStatusesLock.Lock()
18     // send the pod status below
19     s.podStatusChannel <- true
20     s.podStatusesLock.Unlock()
21 }
22
23 func main(){
24     s := &statusManager{podStatusChannel:
       make(chan bool)}      // unbuffered
25     go s.Start()          // G1
26     go s.SetPodStatus()   // G2
27     go s.SetPodStatus()   // G3
28 }
```

Fig. 1: A Mixed Deadlock Concurrency Bug in *Kubernetes*. The Misuse of Channel with the Mutex Leads to the Deadlock.

Unfortunately, existing testing techniques are ineffective in discovering concurrency bugs in Go programs. We first introduce a real example, and then elaborate on how existing techniques are limited. Fig. 1 shows a deadlock bug from *Kubernetes* [1] caused by the misuse of *channel* and *lock* (the code is simplified for clarity). Specifically, it implements a function to manage the status of `Pods`. The code first defines a type (similar to structure/class in C/C++) of `statusManager`, which contains two key methods, `Start` and `SetPodStatus`. The two methods communicate through a shared unbuffered channel (`podStatusChannel`) and mutex (`podStatusLock`) (see such concurrency semantics in Section II-A).

The problem with the code is, with special concurrency execution orders, i.e., interleaving, the thread G_3 created at L_{27} can hold the lock at L_{17} while waiting for the data sent

† Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab.
* Corresponding author

to the unbuffered channel at L_{19}. However, the thread G_1 created at L_{25} which can receive from the unbuffered channel is blocking at L_{10} since it tries to acquire the lock at the same time. Consequently, a deadlock happens, resulting in the non-termination of both G_1 and G_3 as well as resource leaks (see Section II-B for more detail).

For the conventional fuzzing techniques [4], [5], they are limited since merely generating program inputs (e.g., input from *stdin*, *network* or *files*) cannot trigger this bug. Instead, uncovering the deadlock requires enforcing specific interleaving. Several previous works have also proposed methods to search the interleaving spaces to detect concurrency bugs (e.g., the concurrency bug detection tools designed for other languages such as C/C++ [6], [7], Java [8], [9] and C# [10]). However, they cannot be directly applied to detect Go concurrency bugs effectively. First, they are often designed to detect memory access issues such as data races instead of channel-related problems that we are concerned about. Second, they cannot explore the interleaving of channel-related primitives in Go. There are few testing approaches specifically designed for Go targeting on the channel-related bugs. For instance, GoAT [11] utilizes random delay injection before all channel-related operations to explore diverse interleavings. However, recent works [12], [13] show that the random strategy is ineffective because it is hard to cover specific interleavings. The state-of-the-art fuzzing approach for Go, GFuzz [14], demonstrates promising performance by performing directional scheduling via message reordering. However, it only explores the interleavings based on the `select` statements, thus may missing important concurrency bugs that are related to other Go primitives.

To solve these problems, we propose a novel scheduling approach to explore interleavings for Go effectively. In particular, it aims to achieve ***directional*** (cover specific interleavings each time), ***general*** (supporting multiple primitives), and ***fine-grained*** (at the operation-level) scheduling to detect concurrency bugs for Go. However, effectively achieving such a goal poses the following significant challenges.

First, *systematically scheduling the orders of operations on diverse primitives is non-trivial*. GoAT [11] utilizes bounded random delay injection [15] before all channel-related operations to explore diverse interleavings. With the specific focus on the code element of `select`, GFuzz can explore the interleaving space via reordering the concurrency-related operations inside `select` effectively. However, the scheduling method of GFuzz cannot be generalized to other primitives such as `sync.Mutex`. To address the challenge of precisely scheduling the orders of diverse concurrent primitives in Go, we integrate the idea of exploring interleavings via conventional controlled concurrency testing (e.g., [16], [17], [18], [19]) with Go's unique features and propose a new scheduling method driven by *scheduling chains*. Scheduling chains are constrained by primitives and generated based on certain heuristic rules. Different scheduling chains indicate diverse directions to explore new interleavings. The scheduling component instrumented into the concurrent programs will force the execution to follow target directions.

Second, *recognizing scheduling sites and generating effective scheduling chains (see Section III-A) are challenging*. With certain domain knowledge, we can identify suitable scheduling chains for small concurrent programs efficiently, but it is impractical for large-scale open-source programs such as *k8s* and *etcd*. Exploring exhaustively is infeasible for two main reasons. First, the number of interleavings among different primitives can increase exponentially with the increase of primitive operations. Second, the interleavings contain a high degree of duplication, and therefore exploring them exhaustively can waste time and resources, thus being ineffective [12], [13], [20]. To address such a challenge, we propose a novel ***feedback-guided*** testing approach to generate effective scheduling chains. The insight of scheduling chain generation is to learn from both the valueless interleavings and the opportunity of finding new interleavings from existing execution histories. Once the knowledge has been gained, we utilize several ***primitive-constrained*** heuristics to infer and explore new interleavings.

We implement our idea as a tool, `GoPie`, which can achieve effective concurrency testing for **Go** via **P**rimitive-constrained **I**nterleaving **E**xploration. We performed experiments on the Go-specific concurrency benchmark, GoBench [1], as well as well-known real-world open-source Go projects. Via extensive evaluations, we found that `GoPie` is effective and scalable, which can detect more types of concurrency bugs on the GoBench than the existing state-of-the-art baselines. Besides, it has identified 11 unique previous unknown bugs in the open-source projects, 9 of them have been confirmed and 7 of them have been fixed. In addition, the ablation study shows that each component of our design in `GoPie` contributes significantly to its promising performance.

In summary, the main contributions of this paper are:

- ***Originality:*** We provide a new method that can perform directional, general, and fine-grained scheduling supporting multiple primitives for Go concurrency bug detection.
- ***Effectiveness:*** We conducted extensive experiments on the benchmark, GoBench, as well as large-scale open-source projects. The results show that our approach is effective and can outperform the state-of-the-art baselines.
- ***Usefulness:*** Our tool has uncovered 19 previously unknown blocking instances in large-scale open source projects. We filed such blockings as 11 unique bug reports, among which 9 have been confirmed and 7 have been fixed.
- ***Artifact:*** We implemented our tool as `GoPie`. The tool as well as the experimental data is available at https://github.com/CGCL-codes/GoPie to facilitate future studies.

II. Background

A. Concurrency Primitives of Golang

In Go, ***concurrency primitives*** are language features that enable concurrent programming. The following describes some of the most commonly used concurrency primitives in Go [21]:

Goroutines: Goroutines, or routines, are lightweight threads that are used to enable concurrent execution of code in Go. Unlike threads in other languages, goroutines are not mapped to

operating system threads, which makes them more efficient and lightweight. Goroutines can be created easily by the keyword `go` and can be used to perform concurrent tasks, such as I/O-bound or CPU-bound tasks.

Channels: In the Go programming language, channels are a fundamental construct for communication and synchronization between goroutines. A channel is essentially a typed conduit through which values of a particular type can be sent and received between goroutines. Channels are created using the built-in `make` function, which takes a type as an argument and returns a channel of that type (`ch := make(chan int, buffer_size)`). The `<-` operator is used to send and receive values through a channel. For example, `ch <- 42` send an integer to `ch` and `x := <-ch` will receive it from the same channel. In addition to default unbuffered channels (`buffer_size` is 0), which block until both the sender and receiver are ready, Go also supports buffered channels. Buffered channels have a fixed capacity, and sending to a buffered channel blocks only when the buffer is full while receiving from a buffered channel blocks only when the buffer is empty.

Select: Select is a concurrency construct that enables a goroutine to select between multiple channels and wait for the availability of a specific channel. This can be useful in scenarios where a goroutine needs to wait for the completion of a specific task, but can also handle other tasks at the same time if necessary.

Locks: Locks are used to provide mutual exclusion in concurrent programs. In Go, there are two types of locks: mutual exclusion locks i.e., *Mutex* and read-write locks i.e., *RWMutex*. *Mutex* is used to ensure that only one goroutine can execute a specific section of code at a time, while *RWMutex* allows goroutines to concurrently access for read-only operations, whereas write operations require exclusive access.

There are also other concurrency-related primitives such as *WaitGroup* and *Condition Variable*. These concurrency primitives can be used in combination by developers to build high-performance, concurrent Go programs.

B. Concurrency Bugs in Go

Recent work has shown that although channel is a good design, using channels does not eliminate concurrency bugs, and the popularity of channels has also led to a significant proportion of concurrency issues related to channels [3]. Specifically, a recent study shows that among all blocking bugs, channel-related ones take the majority proportion (41/68) [1]. Consequently, in this paper, we mainly focus on *channel-related bugs*. For example, *channel-related runtime errors* such as sending to a closed channel, *communication bugs* such as deadlock caused by cross-routine communication via channels, and *mixed deadlocks* such as the misuse of channels collectively with other concurrency primitives like locks.

We take the bug in Fig. 1 as an example for illustration. Note that the channel is unbuffered since it is created by calling `make(chan bool)`. The second parameter, the buffer length of the channel, is omitted and defaults to 0, which means that the sending and receiving of the channel are synchronized (see Section II-A). For clarity, we use the term *chan* to refer to the variable `s.podStatusChannel`, and the term *mutex* to refer to the variable `s.podStatusesLock` in this example. Additionally, we denote the operation at line x as L_x.

TABLE I: Primitives and Operations Supported by `GoPie`.

Primitive	Operation	Description
routine	-	start a new thread
mutex	*Lock*	acquire the lock
	Unlock	release the lock
rwmutex	*Lock*	acquire write lock
	Unlock	release write lock
	RLock	acquire read lock
	RUlock	release read lock
channel	*Send*	send a message to the channel
	Recv	receive a message from the channel
	Close	close the channel

A deadlock occurs when enforcing the following execution orders (i.e., *interleaving*):

- G_1 executes up to L_9. At the same time, G_2 acquires the lock at L_{17}, and then executes up to L_{19}.
- G_1 and G_2 synchronise, with G_2 sending a message and G_1 receiving it. Then G_2 releases the lock at L_{20}.
- G_1 is now about to execute L_{10}. However, G_3 executes up to L_{17}, which acquires the lock first.
- Then G_3 is blocked at L_{19}. As the channel is unbuffered, G_3 will block here until G_1 is ready to receive from the channel (L_9).

Now there is a deadlock, since G_1 is trying to acquire the lock at L_{10} which is held by G_3. At the same time, G_3 is waiting for G_1 to execute L_9 in the second loop.

III. Approach

In this Section, we introduce our proposed approach `GoPie`.

A. Preliminaries

Besides considering the Go-specific primitives (i.e., channel, goroutine), we also include several traditional primitives (e.g., locks) for the following reasons. First, as previously mentioned, mixed deadlocks caused by the misuse of channels and locks are also common in Go. Second, we aim to show that our approach is general and can be easily applied to other concurrency primitives. Table I shows the concurrency primitives and the corresponding supported operations.

Fig. 2 shows the overview of `GoPie`. Specifically, it takes the Go source code and the associated inputs/unit tests as inputs and outputs potential concurrency bugs. `GoPie` takes several steps to explore interleavings of the inputted Go programs. First, it initializes the *scheduling chain* based on the inputs, which is utilized to drive the process of *order control*. `GoPie` will collect feedback, including *constraints* and *interleavings*, to further assist *interleaving mutation*, which can generate new interesting chains to augment the previous ones. The following introduces the details of each step of `GoPie`.

To formalize our idea, we first define the ***primitive-oriented operation***, denoted as $l : \langle r, p, o \rangle$, to record an operation execution in a concurrent program. Specifically, l is the label, r denotes the concerned goroutine, p represents the primitive

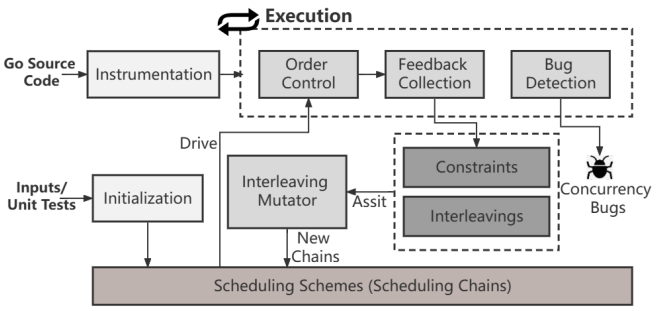

Fig. 2: Overview of GoPie.

(e.g., channel and mutex), and o denotes the operation. For example, G_2 acquiring the lock at L_{17} in Fig. 1 can be denoted as $\langle G_2, mutex, L_{17}\rangle$. The o can be determined statically during instrumentation while r and p are determined dynamically for the determination is non-trivial via static analysis. To avoid confusion, we use the term *primitives* to denote object-like primitives such as channels and locks. Meanwhile, we refer to the special primitive of goroutine as *routine*.

Based on it, we further define an important concept, ***scheduling chain (SC)***, which is used in this study as follows:

$$SC : \{\langle r_1, p_1, o_1\rangle ..., \langle r_i, p_i, o_i\rangle, ..., \langle r_n, p_n, o_n\rangle \mid r_i \neq r_{i+1},\ 1 \leq i \leq n-1\} \quad (1)$$

Each triple in a SC refers to an operation execution of a concurrency primitive. The whole chain stands for the order in which these concurrency-related operations should occur. For example, in Fig. 1, the interleaving that caused blocking requiring G_3 to execute L_{17} before G_1 executing L_{10} at the third step, can be represented as $\{\langle G_3, mutex, L_{17}\rangle, \langle G_1, mutex, L_{10}\rangle\}$.

To ensure that the interleaving is correctly scheduled, SC mandates that two consecutive triples in a chain belong to different threads (i.e., $r_i \neq r_{i+1}$). Such a constraint is also guaranteed at the phase of mutation (see Section III-D). The scheduling chain plays a critical role in driving the order control process to perform directional scheduling (trying to cover the target specified interleavings each time, see Section III-B), and enabling the mutator to be effective even without the modeling of concurrency mechanisms.

B. Order Control

Go-specific testing tool, GFuzz [14], reorders the operations of different channels in the same `select` statements, which is achieved by controlling the program to prioritize the execution of a specific `case` branch in the `select` statement. Unfortunately, GFuzz only schedules the operations in the same `select` statement, which is a non-trivial limitation (see our evaluation in Section IV for more details). In this study, we propose a more general scheduling method on diverse concurrency primitives and operations, which can further serve for our interleaving exploration as introduced in Section III-D, the *scheduling chain*. Our scheduling aims to enforce the program to cover the target interleavings as specified by the scheduling chains without affecting the original program logic. To achieve such a goal, GoPie instruments general controlling

Algorithm 1: General Stub around Concurrency Operations

```
1  Data: active_set, wait_queue ← ParseSchedChain()
   Input: the unique operation op next to the stub.
   Output: the result of scheduling res, initialized to NULL.
   /* Bailing Out                                          */
2  if op ∉ active_set then
3      ExecuteOperation(op) ;
4      res ← StubNotActive ;
   /* Stub Before Executing op                             */
5  while res ≠ NULL and op ≠ GetFirst(wait_queue) do
6      if IsTimeout() then
7          res ← FailedBecauseTimeout ;
8          break ;
9      if IsCancelled() then
10         res ← FailedBecauseCancelled ;
11         break ;
12     TransitorySleep() ;
   /* Executing op                                         */
13 ExecuteOperation(op) ;
   /* Stub After Executing op                              */
14 Dequeue(wait_queue, op) ;
15 return res == NULL ? Success : res
```

stubs around each concurrency operation of various primitives and orchestrates such stubs to work collectively. For instance, the stubs will wrap L_{17}, L_{19}, L_{20} in function `SetPodStatus` in Fig. 1 separately. Algorithm 1 shows the instrumented stub around the concerned operations. For clarity, we use A_x to denote the line number in the algorithm (e.g., A_{13} for the execution of concerned operations). The algorithm contains the following main steps.

First, *initialization* (A_1). Before each controlled execution, the corresponding scheduling chain will first be parsed into two global structures, $active\_set$ and $wait\_queue$ respectively to record the active stubs and the sequence.

Second, *bailing out* (A_2 to A_4). For those inactive stubs, the algorithm ignores the stub code around the original operations, thereby reducing the stubs' overhead as much as possible.

Third, *roll polling* (A_5 to A_{12}). The stub blocks the execution of the current routine by roll polling. The blocking takes effect until the corresponding operation becomes the first operation in the queue. In other words, the roll polling mechanism ensures that each operation is executed in the order as specified in the scheduling chain, ***with all the preceding operations completed before the next one starts.***

Finally, *updating* $wait\_queue$. After the original operation has been executed, the $wait\_queue$ will be updated at A_{14}. Consequently, all the cooperative stubs can recheck whether the operation can now be executed at A_5.

In the example as mentioned at Section III-A, if G_1 is about to execute L_{10} before G_3 executing L_{17}, G_1 will be blocked at the roll polling until G_3 finishes its work and updates the queue at A_{14} of Algorithm 1 in the stub.

The following points should be noted in Algorithm 1. First, if the interleaving specified by the scheduling chain cannot be achieved, the stub will give up blocking due to timeout (A_8).

Second, when the main thread is about to terminate and the blocking detector is about to work, all the blocking routines will cancel the roll polling (A_{11}) to avoid blocking on these stubs, which may cause false positives. Lastly, the scheduling results will be collected and used as runtime feedback.

C. Feedback Collection

`GoPie` collects runtime information as feedback to guide the exploration of interleavings instead of merely measuring the effect of scheduling. In particular, `GoPie` learns from the interleaving of previous executions to infer new potential interleavings (see Section III-D).

Building on recent advancements in static analysis for Go [22], [23] and the edge coverage achieved through conventional fuzzing, we propose a novel runtime feedback mechanism for interleavings. Specifically designed to aid the exploration of Go-specific concurrent primitives, we denote this new mechanism as $CPOP$ (Concurrency Primitive Operation Pair).

$CPOP$ refers to the execution order of concurrency-related operations, which can be defined as a pair of operations (i.e., $\langle op_1, op_2 \rangle$), where the operation is defined in Section III-A. During interleaving mutation, both the interleavings inside the same routine and those among different routines are important. Therefore, we record two types of $CPOP$ generated by a single execution: *interleaving pairs within the same routine* (denoted as $CPOP_1$) and *interleaving pairs among different routines* (denoted as $CPOP_2$). The following introduces the details.

$CPOP_1$, whose operations are from the same routine, is similar to the traditional edge coverage with limited statements concerned (i.e., those concerned primitives). However, in the concurrent scenario, $CPOP_1$ is not only affected by the program inputs but also related to the interleaving of routines. For example, as shown in Fig. 1, $\{\langle G_2, mutex, L_{17} \rangle, \langle G_2, chan, L_{19} \rangle\}$ belongs to $CPOP_1$ concerning G_2.

$CPOP_2$, whose operations are in different routines but have the same primitive, is used to measure the interleavings among different routines. In this case, we use primitives as a constraint, as they provide a clear and direct relationship between two operations. For instance, in the example as shown in Fig. 1, $\{\langle G_3, mutex, L_{20} \rangle, \langle G_1, mutex, L_{10} \rangle\}$ belongs to $CPOP_2$ from G_3 to G_1. Furthermore, the connection between different primitives is also considered subsequently in Section III-D.

Both $CPOP_1$ and $CPOP_2$ are important to aid the process of interleaving mutation.

D. Interleaving Mutator

We have introduced *how* to perform scheduling as discussed above while there is another key question, that is *what* to schedule. In this Section, we introduce how `GoPie` performs mutation to generate more valuable scheduling chains guided by the feedback collected in Section III-C. The insight of `GoPie`'s mutation strategy is to infer potential useful scheduling chains based on previous knowledge, the effectiveness of which will further be dynamically confirmed via specific order control as introduced in Section III-B. During the loop between new potential interleavings inferred and the interesting interleavings confirmed, the concurrency program is explored by diverse interleavings and concurrency bugs can be triggered at the same time. However, there are plenty of different primitives and operations, but not all of them are valuable to be scheduled for the sake of both efficiency and effectiveness. The following introduces our strategy to infer new potential interleavings during interleaving mutation.

1) Preliminary Selection: In Go, routine switching only occurs at locations where blocking is possible, such as *channel operations*, *locks*, etc. First, as described in Algorithm 1, we only consider the concurrency-related locations as the possible scheduling sites and perform instrumentation around such operations during static analysis. Other statements are ignored, e.g., L_{11} and L_{18} in Fig. 1. Moreover, in order to avoid the huge overheads of scheduling all possible sites in the program, only those sites in the scheduling chain are activated in order to control the execution (see `active_map` in Section III-B).

2) Primitive-oriented Constraint: Recent static analysis approaches for Go [22], [23], [24], have employed the strategy of dividing programs into smaller fragments to avoid the path explosion problem in concurrent scenarios. In particular, these techniques have introduced several rules to determine whether two operations are related and then categorize unrelated primitives into separate sets via static analysis. A localized analysis is then conducted on each set. Inspired by this idea, we propose to divide the interleaving space based on the involved primitives and the routines they are related to. Specifically, we design the following rules.

$$
\begin{aligned}
&Rule1: \exists c, c', \langle c, c' \rangle \in CPOP_1 \rightarrow c' \in Rel_1(c) \\
&Rule2: \exists c, c', \langle c, c' \rangle \in CPOP_2 \rightarrow c' \in Rel_2(c) \\
&Rule3: \exists c, c', c'', c' \in Rel_1(c), c'' \in Rel_2(c') \rightarrow c'' \in Rel_2(c) \\
&Rule4: \exists c, c', c'', c' \in Rel_2(c), c'' \in Rel_2(c') \rightarrow c'' \in Rel_2(c)
\end{aligned}
\tag{2}
$$

where $Rel_{1/2}(x)$ stands for the set in which the operations of primitives are related to x. Rule 1 indicates that in one execution, two operations executed consecutively in the same routine are related. In other words, two operations that can constitute a $CPOP_1$ edge, are marked as $CPOP_1$ related (i.e., Rel_1). Similarly, Rule 2 indicates that two operations of the same primitive that execute consecutively in different routines are related. That is, two operations that can constitute a $CPOP_2$ edge, are considered as $CPOP_2$ related (i.e., Rel_2).

Rule 3 and Rule 4 perform transitive inferences. Specifically, Rule 3 performs a different transitive inference from Rel_1 to Rel_2. The main insight of Rule 3 is to explore more cross-routine interleavings by sliding on the intra-routine interleavings of each routine. For example, since $\langle G_2, mutex, L_{20} \rangle \in Rel_1(\langle G_2, chan, L_{19} \rangle)$ based on Rule 1 and $\langle G_3, mutex, L_{17} \rangle \in Rel_2(\langle G_2, mutex, L_{20} \rangle)$ based on Rule 2, we can infer based on Rule 3 that $\langle G_3, chan, L_{17} \rangle \in Rel_2(\langle G_2, mutex, L_{19} \rangle)$. Rule 4 is straightforward, which shows that the relationship inside a set of Rel_2 is transitive. For example, supposing c and c', c' and c'' respectively operate on the same primitive but in different routines, there is a high possibility that c and c'' are also located at different routines and operate on the same primitive. By adopting the

fragment technique from static analysis and integrating it with the primitive-constrained feedback, GoPie can effectively partition program operations, thus minimizing the likelihood of generating extraneous scheduling. In the simplified example as shown in Fig. 1, all the concurrent operations operate on the *channel*, and the *mutex* are related based on the above rules, which can isolate them from other concurrent operations in the original program that is not simplified.

3) Potential Interleaving Mutation: In this Section, we introduce how to infer potential interleavings that were not covered before based on the feedback gained from previous executions. Specifically, GoPie learns whether a scheduling chain should be further mutated from the following aspects: 1) *whether the scheduling chain was actually scheduled*. As mentioned in Section III-B, GoPie can infer whether the current chain indeed takes effect from the runtime feedback. If not, GoPie will not mutate it as it is less likely to generate new valid interleavings and trigger bugs. 2) *whether timeout occurs during testing*. Some tests require a long-term execution (e.g., over 7 minutes), which is inefficient and will cause huge overheads. Therefore, we ignore such cases for the sake of efficiency. For the remaining scheduling chains, GoPie adopts the following four mutation operators in conjunction with the learned knowledge, Rel, as introduced above.

1. Abridge: it removes an item from the SC (either from head or tail) if there is more than one operation in it, which helps GoPie to limit the length of an SC.

$$\begin{aligned} &\exists o_i, o_j, \{o_i, o_{i+1}..., o_{j-1}, o_j\} \in SC \\ \rightarrow &\{o_{i+1}, ..., o_{j-1}, o_j\}, \{o_i, o_{i+1}..., o_{j-1}\} \in SC \end{aligned} \tag{3}$$

2. Flip: it performs a reverse process on the SC to be mutated. For example, if $\langle s_1, s_2 \rangle$ is covered in the previous scheduling, $\langle s_2, s_1 \rangle$ is also valuable to take a try. In this case, the flip operation works.

$$\begin{aligned} &\exists o_i, o_j, \{..., o_i, o_j, ...\} \in SC \\ &\rightarrow \{..., o_j, o_i, ...\} \in SC \end{aligned} \tag{4}$$

3. Substitute: it tries to replace an operation with another one from the set of Rel_1 (which can be found and collected from feedback in previous executions). For example, if $\langle s_1, s_2 \rangle$ is covered, GoPie tries to explore the interleaving of the intra-routine predecessor and the succeeding operation of s_2.

$$\begin{aligned} &\exists o_j, o_j \in Rel_1(o_i), \{..., o_i, ...\} \in SC \\ &\rightarrow \{..., o_j, ...\} \in SC \end{aligned} \tag{5}$$

4. Augment: it tries to increase the length of SC by adding another operation from the set of Rel_2 to its tail, which aims to explore those effective interleavings in a further step.

$$\begin{aligned} &\exists o_j, o_j \in Rel_2(o_i), \{..., o_i\} \in SC \\ &\rightarrow \{..., o_i, o_j\} \in SC \end{aligned} \tag{6}$$

For better understanding, we provide an example of mutating an existing interleaving to trigger the bug in Fig. 1. Fig. 3 shows a normal execution and buggy execution of the example. The key interleavings to trigger this deadlock are denoted as the red edges in the figure. Specifically, the deadlock bug can be triggered by the scheduling chain of

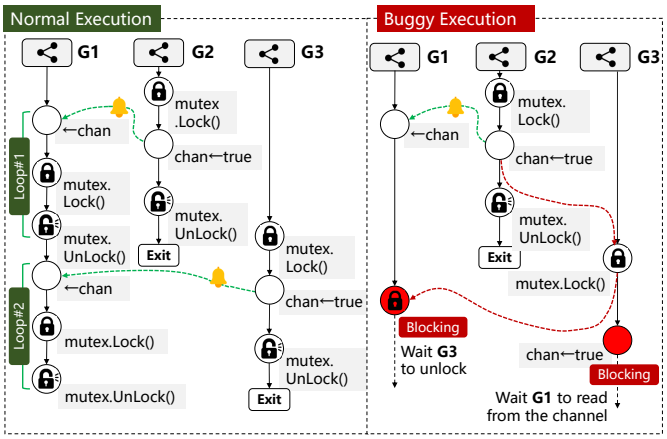

Fig. 3: Different Interleavings of Normal and Buggy Executions for the Example as Shown in Fig. 1.

TABLE II: Interleavings Collected during the Normal Execution of the Example in Fig. 1, where $step_x$ Denotes that the CPOP is Utilized in which Mutation Step as Shown in Table III.

Type	ID	$CPOP_1$ (Rel_1)
G_1	1	$\langle\langle G_1, chan, L_9\rangle, \langle G_1, mutex, L_{10}\rangle\rangle$
	2	$\langle\langle G_1, mutex, L_{10}\rangle, \langle G_1, mutex, L_{12}\rangle\rangle_{Step_2, Step_5}$
	3	$\langle\langle G_1, mutex, L_{12}\rangle, \langle G_1, chan, L_9\rangle\rangle$
G_2	4	$\langle\langle G_2, mutex, L_{17}\rangle, \langle G_2, chan, L_{19}\rangle\rangle$
	5	$\langle\langle G_2, chan, L_{19}\rangle, \langle G_2, mutex, L_{20}\rangle\rangle_{Step_1}$
G_3	6	$\langle\langle G_3, mutex, L_{17}\rangle, \langle G_3, chan, L_{19}\rangle\rangle$
	7	$\langle\langle G_3, chan, L_{19}\rangle, \langle G_3, mutex, L_{20}\rangle\rangle$
		$CPOP_2$ (Rel_2)
chan	8	$\langle\langle G_1, chan, L_9\rangle, \langle G_2, chan, L_{19}\rangle\rangle$
	9	$\langle\langle G_2, chan, L_{19}\rangle, \langle G_1, chan, L_9\rangle\rangle$
	10	$\langle\langle G_1, chan, L_9\rangle, \langle G_3 : chan, L_{19}\rangle\rangle$
mutex	11	$\langle\langle G_2, mutex, L_{20}\rangle, \langle G_1, mutex, L_{10}\rangle\rangle_{Initial}$
	12	$\langle\langle G_1, mutex, L_{12}\rangle, \langle G_3, mutex, L_{17}\rangle\rangle_{Step_3}$
	13	$\langle\langle G_3, mutex, L_{20}\rangle, \langle G_1 : mutex, L_{10}\rangle\rangle$

$\{\langle G_2, chan, L_{19}\rangle, \langle G_3, mutex, L_{17}\rangle, \langle G_1, mutex, L_{10}\rangle\}$ (following the red edges in Fig. 3). Therefore, the goal of the interleaving mutator is to infer such a triggering scheduling chain. GoPie's insight is to leverage the execution histories to perform mutation. Table II shows some examples of the execution histories, which can be collected during the testing process. For instance, during the normal execution as shown in Fig. 3, the execution of different primitive operations in G_2 follows the orders of: $\langle G_2, mutex, L_{17}\rangle$, $\langle G_2, chan, L_{19}\rangle$ and $\langle G_2, mutex, L_{20}\rangle$ respectively, and thus can form two $CPOP_1$ pairs as shown in Table II. Other pairs as shown in the table can be similarly collected based on execution histories.

Table III shows the details of the mutation procedures as adopted by GoPie to detect this bug. Specifically, in this example, GoPie starts from the existing covered interleaving, $Initial$:$\{\langle G_2, mutex, L_{20}\rangle, \langle G_1, mutex, L_{10}\rangle\}$. For clarity, we have already added some notations as subscripts in Table II to denote that the pair will be used in which mutation. For instance, in $Step_1$, $CPOP$#5 is utilized in the mutator of *Substitute*, thus resulting the chain of $\{\langle G_2, chan, L_{19}\rangle, \langle G_1, mutex, L_{10}\rangle\}$. After a sequence of further mutations of *Substitute*, *Abridge*, *Flip* and *Substitute*,

TABLE III: The Mutation Steps of GoPie for the Example.

Step	Operator	The Generated Scheduling Chain
$Initial$	-	$\{\langle G_2, mutex, L_{20}\rangle, \langle G_1, mutex, L_{10}\rangle\}$
$Step_1$	Substitute	$\{\langle G_2, chan, L_{19}\rangle, \langle G_1, mutex, L_{10}\rangle\}$
$Step_2$	Substitute	$\{\langle G_2, chan, L_{19}\rangle, \langle G_1, mutex, L_{12}\rangle\}$
$Step_3$	Abridge	$\{\langle G_2, chan, L_{19}\rangle, \langle G_1, mutex, L_{12}\rangle, \langle G_3, mutex, L_{17}\rangle\}$
$Step_4$	Flip	$\{\langle G_2, chan, L_{19}\rangle, \langle G_3, mutex, L_{17}\rangle, \langle G_1, mutex, L_{12}\rangle\}$
$Step_5$	Substitute	$\{\langle G_2, chan, L_{19}\rangle, \langle G_3, mutex, L_{17}\rangle, \langle G_1, mutex, L_{10}\rangle\}$

the result of $Step_5$ can generate the target bug-triggering scheduling chain, and thus the bug can be detected by GoPie.

For the sake of clarity, we demonstrate the steps following the order to trigger the bug. In the actual implementation, the order of testing those potential interleavings is randomized (to avoid getting caught in localized search space) [25]. This means that the process of discovering the bug will be more convoluted, and thus appropriate guidance is necessary.

E. Concurrency Bug Detection

As the Go runtime can only detect global deadlocks (i.e., all routines are deadlock) and runtime errors [26], GoPie uses an additional detector named *Goleak* [27] to gain the ability to detect partial blocking. *Goleak* can be used to detect if there is any goroutine that has not been terminated when the main routine is about to exit. Specifically, GoPie starts a timer at the beginning of each execution, and the leak checking is performed when either the execution is about to terminate or the timer is timeout. GoPie then invokes the API of goleak which will perform blocking checks, to ensure all runnable goroutines have sufficient time to finish their task and quit normally. Finally, if there are still routines remained, GoPie reports a warning of the blocking situation. Before performing the leak checking, a *cancel* signal will be broadcasted to all goroutines to avoid false positives caused by order control (see Section III-B).

GoPie are expected to output the location of blockings, the scheduling chain, and the logs of program execution. For example, the report of the bug in Fig. 3 will include $\langle G_1, mutex, L_{10}\rangle$, $\langle G_3, channel, L_{19}\rangle$ as the blocking location, $\{\langle G_2, chan, L_{19}\rangle, \langle G_3, mutex, L_{17}\rangle, \langle G_1, mutex, L_{10}\rangle\}$ as the key interleaving and logs produced by the tested program. We utilize this information to provide explanations for the occurrence of blockings, as well as recommendations for resolving them. Eventually, we report the results of our analysis, the location of the blocking, and the possible fixes to the developer via pull requests.

To avoid bothering developers with too many imprecise bug reports and alleviating their workload, we further manually analyzed all the blockings detected by GoPie before filing bug reports. There are two main kinds of blocking instances that have been excluded from reporting. First, there are a few benign blockings detected in practice. These blockings are true positives but are acceptable to developers. The information of such blockings can often be found in the comments of code or Github issues [28]. GoPie employs a short whitelist of such expected leaked goroutines to filter them out since the developers have already explained that they are acceptable. Second, there are few false positives caused by the overhead theoretically, which require a manual review. Due to efficiency considerations, we cannot afford to wait indefinitely for goroutines to complete their execution. For instance, if a goroutine calls *time.Sleep()* for an extended period and continues to obstruct after the main routine has terminated, it will be identified as blocked.

GoPie has identified 19 instances of blocking after filtering, as detailed in Section IV-B. We have merged those similar blocking instances into one bug report. Ultimately, we have submitted a total of 11 reports to developers, some of which contain multiple instances of blocking.

IV. Empirical Evaluation

We implement GoPie based on Golang. The instrumentation is achieved based on the Token package and AST package [29] and the concurrency operation trace is implemented by instrumenting the runtime of Golang 1.19. Besides, We also reuse the source code of *goleak* to detect partial blocking bugs. Our implementation contains 7,163 lines of Go source code and Shell scripts. All our experiments are performed on a server with Intel(R) Xeon(R) Gold 6248R CPU and 128GB RAM.

To evaluate the effectiveness and usefulness of GoPie, we aim to answer the following three questions:

- **RQ1-Effectiveness:** How effective is GoPie in bug detection on the benchmark?
- **RQ2-Usefulness:** Can GoPie detect real bugs that have not been exposed?
- **RQ3-Ablation Study:** How does each component of GoPie contribute to its effectiveness?

A. RQ1: Effectiveness

1) Methodology: In this RQ, we evaluate GoPie's effectiveness in terms of its capability to detect existing Go concurrency bugs. We select GFuzz [14] as our baseline since it is the present state-of-the-art dynamic concurrency testing tool for the Go concurrency bug.

In particular, we evaluate it on the benchmark dataset GoBench [1]. GoBench is designed to assist researchers in comprehending concurrency bugs in Go and fairly comparing them among different detection tools. There are 82 real bugs from 9 popular open-source applications (GoReal) and 103 bug kernels (GoKer) in GoBench. For each real-world bug in GoReal, GoKer has extracted and simplified the bug-relevant code while preserving the bug-inducing logic, including the root cause and the triggering interleavings (and thus denoted as ***bug kernels***). We use the blocking bug kernels in GoKer (68 bug kernels extracted from 40 real-world bugs) for evaluation in this RQ since the remaining ones are mainly data race related, which is out of the scope of this study. We only evaluate GoPie and GFuzz on GoKer in this RQ since we failed to run GFuzz on the dockers of GoReal as provided on the official page. Besides, both GoPie and GFuzz utilize the unit tests associated with the projects as the testing entry, and thus we also utilize the ***unit test*** as a baseline to examine whether the bugs in GoKer can be easily detected by the repeating execution of unit tests without scheduling as adopted by GoPie.

TABLE IV: The Effectiveness of GoPie on GoBench. The column of *Bugs* is the number of bug kernels in each project, and $select$, op_{chan}, op_{lock} stand for the number of $select$ statements, channel related statements outside the scope of $select$, as well as the lock-related statements respectively.

Benchmark Information					Dynamic Tools		
Project	Bugs	$select$	op_{chan}	op_{lock}	unit test	GFuzz	GoPie
cockroach	18	11	28	18	7	2	18
etcd	9	4	30	20	4	1	8
grpc	7	6	27	12	1	4	7
hugo	2	1	2	2	2	0	1
istio	3	5	11	5	1	2	3
kubernetes	13	14	47	39	3	4	13
moby	13	9	31	19	2	5	13
serving	1	1	8	3	0	0	1
syncthing	2	2	3	2	2	0	2
Total	68	53	187	120	22	18	**66**

TABLE V: The Effectiveness of GoPie on Channel-related Bugs. *Total* denotes the number of bugs in each root cause. CV: Condition Variable; C: Context; L: Lock; W: WaitGroup

Benchmark Information					Dynamic Tools		
Root Cause	Total	$select$	op_{chan}	op_{lock}	unit test	GFuzz	GoPie
Channel	17	20	61	9	6	9	17
Channel & CV	2	8	11	7	1	1	2
Channel & C	7	9	26	2	0	4	7
Channel & L	13	9	63	42	3	4	13
Channel & W	2	2	6	7	2	0	2
Total	41	48	167	67	12	18	**41**

For GFuzz, we use the default fuzzing configurations as adjusted by the authors and we run GoPie and the baselines for 12 hours with the same number of threads.

2) Result: Table IV shows the evaluation results. In total, GoPie successfully conducted testing on 67 of the 68 bug kernels and detected 66 of them (98.51%), while the baseline can only detect at most 18 out of the 65 successfully compiled kernels (27.70%) among different configurations. The unit tests in GoKer have detected 22 global deadlock bugs. To gain a deeper understanding of the underlying reasons, we conducted a lightweight static analysis to determine the number of different primitives utilized in each project (as shown in Table IV). Specifically, $select$ denotes the number of `select` statements, which is the main focus of GFuzz; op_{chan} denotes channel-related statements that are outside the scope of `select` statements; and op_{lock} denotes lock-related statements (all such lock statements will not appear in any `select` statement). Both op_{chan} and op_{lock} can be scheduled by GoPie while being ignored by GFuzz. As we can see in Table IV, around 85% (i.e., 1 - 53/(53+187+120)) of the operations cannot be handled through `select` statements, thus being missed by GFuzz. On the contrary, they can be effectively handled by GoPie since it supports multiple concurrent primitives in Go.

To further demonstrate the effectiveness of GoPie, we inspect the detection results only in terms of *channel* related bugs (including *channel-misued* ones and those mixed with other primitives, such as *lock*, *context*). Table V shows the statistical results, which indicates that GoPie can detect all the *channe-related* bugs in the benchmark, while the performance of GFuzz is still unsatisfying. GFuzz failed to detect many of them due to the inadequate scheduling method. On the contrary, GoPie schedules more primitive sites and employs our novel designs to effectively explore interleavings, thus resulting in more bugs detected.

TABLE VI: Open Source Projects in RQ2. The *KLOC* column shows the number of code lines in each project; The *Stars* column indicates the stars it has received on GitHub.

Name	KLOC	Description	Stars
kubernetes	3,453	Container Scheduling and Management for Production	98K
prometheus	1,186	Monitoring system and time series database.	48K
etcd	181	Distributed reliable key-value store	43K
go-ethereum	368	Ethereum protocol written in Go	42K
tidb	476	Open-source, cloud-native, distributed database	34K
grpc-go	117	The Go language implementation of gRPC.	18K

Answer to RQ1: GoPie is effective, which can detect most of the concurrent bugs in GoKer of GoBench. It can also outperform the state-of-the-art baseline significantly.

B. RQ2: Usefulness

1) Methodology: In this RQ, we evaluate GoPie's usefulness in examining whether it can detect real bugs in large open-source projects that are previously unknown. Specifically, we run GoPie on the latest version of the projects as shown in Table VI that are used in previous works [14], [23]. We select these projects for the following reasons: First, projects in this benchmark are the most widely used ones by existing studies and they have also employed lots of Go concurrency primitives. Second, existing studies [14], [22], [23] have already sufficiently tested these projects, and thus it is also more challenging to discover new bugs. Therefore, the detection of new bugs can enhance the usefulness of our approach. Similar to GFuzz, GoPie utilizes all the unit tests in the projects as the fuzzing entry, and each project is tested for 12 hours.

For those bugs GoPie detected, we also try to examine whether GFuzz can detect them. Besides, we also evaluate the reported bugs by executing the unit test provided in the projects with the same bug detector, GoLeak, in our implementation (see Section III-E), to check out whether these bugs can also be detected by the approach without scheduling.

2) Result: In total, GoPie has detected 19 concurrency-blocking instances in four out of the six selected projects. We have checked all the reports manually and finally filed 11 bug reports to the maintainers with 9 of them being confirmed. We only filed 11 bug reports since that for several blockings, although they are discovered from different entries (unit test), they are duplicated for having the same root cause. Therefore, we submit them collectively in one report. We have also helped to fix 7 out of the 9 confirmed bugs by crafting patches and making pull requests. Table VII shows all the reported unique concurrency bugs. As we can see, both merely considering the unit tests without scheduling and GFuzz cannot detect any of the bugs. There are two main reasons for GFuzz's ineffectiveness. First, some bugs are detected under diverse concurrent primitives that are out of the scope of GFuzz. Second, GFuzz failed to compile the latest versions of certain

TABLE VII: Bug Reports Filed by GoPie. '-' indicates that the issue or pull request is still waiting for responses.

Project	Bug	Status			Baseline	
		reported	confirmed	fixed	GoLeak	GFuzz
grpc	grpc_6190 [30]	✓	✓	✓	✗	✗
	grpc_6174 [31]	✓	✓	✓	✗	✗
	grpc_6172 [32]	✓	✓	✓	✗	✗
etcd	etcd_15785 [33]	✓	✓	✓	✗	✗
	etcd_15675 [34]	✓	✓	✓	✗	✗
	etcd_15723 [35]	✓	-	-	✗	✗
kubernetes	k8s_117517 [36]	✓	✓	-	✗	✗
	k8s_117534 [37]	✓	✓	✓	✗	✗
	k8s_114071 [38]	✓	✓	✓	✗	✗
eth	eth_27675 [39]	✓	✓	-	✗	✗
	eth_27110 [40]	✓	-	-	✗	✗

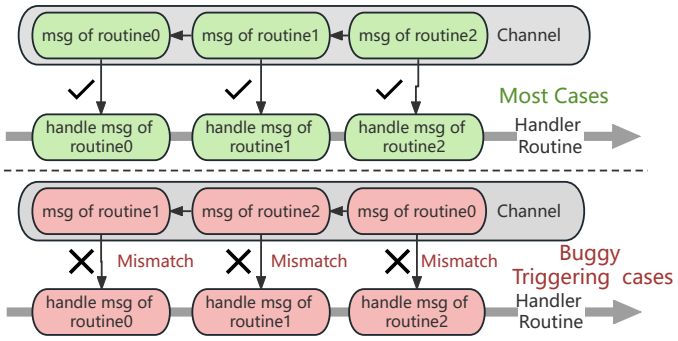

Fig. 4: A Previously Unknown Concurrency Bug in *etcd* (i.e., *etcd_15675*) Detected by GoPie.

projects due to the new language features used in such projects even though we have followed the guidance and fuzzing strategies as shown on the official page of GFuzz.

3) Case Study: Fig. 4 shows a previously unknown bug in project *etcd* detected by GoPie (i.e., etcd_15675), which is caused by the wrong assumption of execution orders. In this bug, several routines (message provider routines) share one channel to send messages to another routine (message handler routine). The handler routine processes the message with a fixed sequence since the developers expect the message provider routines to send messages following the same sequence as how the handler routines process the messages. The issue here is that the sequence of message arrival is not guaranteed but can be affected by time delay caused by `time.After` or computation load. It works normally for most cases but can lead to unexpected errors in certain situations. Unfortunately, the buggy triggering case is hard to cover by random delay injection or insufficient interleaving exploration.

GoPie detected the hidden bug during interleaving exploration via controlled scheduling instead of injecting random delay. We reported it and made a pull request to fix it by using separate channels to handle the messages from different routines, which is confirmed and approved by the maintainers.

Answer to RQ2: GoPie is useful, which can detect previous unknown concurrent bugs for large-scale Go projects that cannot be detected by the existing baselines.

C. RQ3: Ablation Study

1) Methodology: In this RQ, we examine the contribution of the major components of GoPie, including the *feedback guidance* and *interleaving mutation*. In particular, we disable one component each time and perform fuzzing for 48 hours to compare GoPie's performance. The two designed variants are GoPie-FB, and GoPie-Mu, which denote GoPie without feedback guidance and without interleaving mutation respectively. GoPie-FB performs interleaving mutation without the primitive constraints learned from feedback and existing covered interleavings. GoPie-Mu will always use an empty chain, and do not perform scheduling during testing.

Since one important metric we compare is the number of detected blockings and the unique bugs, we perform the ablation study on all the packages of the projects used in RQ2 that have bugs detected by GoPie (eight different packages in total). Be noted that we do not perform fuzzing on the whole project for the sake of efficiency.

Besides the bug-finding ability, we also evaluate from other aspects in order to understand the behind effectiveness, including *concurrency coverage*, *the number of unique effective scheduling chains*, and *the number of critical channel states*. First, for concurrency coverage, we use the coverage feedback as defined in Section III-C since it can reflect the interleavings between executions. Specifically, we use the operation pairs executed continuously within the same routine ($CPOP_1$) and across different routines ($CPOP_2$) to investigate the interleavings. The higher the interleavings, the more opportunities for triggering a concurrency bug. Second, for the number of generated unique effective scheduling chains, we can distinguish whether a scheduling chain is effective based on the feedback (e.g., whether a chain performs correct scheduling during execution). The larger the number, the more effective the interleaving exploration process. Third, we also consider the state of the channels since recent works have noticed that in Go programs, only the channels in certain states can blocking occurs [23], [24], [41]. For instance, if a channel is closed, a new arrival send operation will cause a '*send to closed channel*' panic in the program, which will further lead to a crash. GFuzz utilizes certain special states, such as *channel full*, *channel empty*, and *channel close* as the feedback, and considers them as critical states [14]. Consequently, we utilize the number of such critical states as a metric since it can reflect the likelihood of triggering channel-related bugs.

2) Result: Table VIII shows the results of *blockings and bugs detected*. It shows that GoPie can detect a total of all 19 blockings in 11 unique bugs. However, without scheduling, GoPie can only detect 3 out of the 11 unique bugs. Equipped with scheduling but without the feedback (GoPie-FB), it only achieved one more blocking without any new bug detected compared to the group without scheduling (GoPie-Mu). The performance reduction indicates that both the scheduling and the feedback guidance contribute significantly to the performance of GoPie.

Table IX shows the results in terms of the other metrics,

TABLE VIII: Bug Detection Results of GoPie and the Variants. In X (Y): X refers to the number of blockings while Y refers to the number of unique bugs. '-' means no bug detected.

Package	Blockings (Unique Bugs)		
Group	GoPie	GoPie-FB	GoPie-Mu
go-ethereum/console	5 (1)	-	-
go-ethereum/eth/fetcher	2 (1)	2 (1)	2 (1)
grpc-go/stats/opencensus	1 (1)	-	-
grpc-go/xds	2 (2)	-	-
etcd/server/etcdserver	1 (1)	-	-
etcd/tests/integration/snapshot	3 (1)	1 (1)	-
etcd/tests/integration/proxy/grpcproxy	1 (1)	-	-
k8s.io/client_go/tools/watch	4 (3)	1 (1)	1 (1)
Total	19 (11)	4 (3)	3 (3)

which can explain the behind effectiveness. We can observe that GoPie achieves the optimum performance for most cases while its performance is degraded either disabling the feedback guidance or interleaving mutation. Specifically, with respect to concurrent coverage, GoPie can cover 16,483 and 18,367 pairs for $interleaving_1$ and $interleaving_2$ respectively. However, without feedback, the number has dropped by 6.2% and 6.4% respectively; without interleaving mutation, the number has dropped by 8.2% and 9.3% respectively. With respect to the generated valid scheduling chain, the performance has been degraded more significantly. In particular, GoPie can generate over 10 times more valid scheduling chains compared with the other two variants. Similar results have also been observed with respect to the critical channel state. Such results demonstrate the effectiveness of our feedback-driven scheduling in interleaving exploration, which can explain the promising performance achieved by GoPie.

Answer to RQ3: The major design of GoPie contributed significantly to its performance. Specifically, performing scheduling driven by execution feedback and interleaving mutation is promising for detecting Go concurrency bugs.

V. Discussion and Future Work

This study is limited in the following aspects.

First, we only implement the scheduling on channel-related and lock-related operations to demonstrate GoPie's effectiveness. However, there are other concurrency primitives such as *WaitGroup*, and *Cond*. The misuse of these primitives can also lead to concurrency bugs. Fortunately, our approach can easily be applied to these primitives, and thus is potential to identify more interleavings and more types of concurrency bugs.

Second, GoPie does not mutate the provided test case of each program, and thus it will miss those interleavings that require extra test cases to cover. There are many previous works focus on test case generation such as input-driven fuzzing [4], [5], [42] and unit test generation [43], [44], [45], [46]. GoPie can be integrated with such techniques to further enhance its effectiveness. It is also a promising direction worth exploring in the future.

Third, there are both potential false positives and negatives in GoPie. The implementation of GoPie made a trade-off between efficiency and completeness, which can result in false negatives. In particular, we set timeouts to the processes of order control, and unit tests that execute over the time budget will be abandoned. Besides, the length of scheduling chains is also bounded, bugs that require many steps of scheduling to be triggered are not guaranteed to be detected. As mentioned in Section III-E, there are a few false positives caused by the detector due to expected leaks and time limitations of detection. The expected leaks can be filtered out by a short project-specific whitelist (e.g., three in *go-ethereum*). We will explore how to automate this process in the future. Besides, runtime partial blocking detection [47] can also reduce the false positives caused by *Goleak*. We plan to explore how to integrate it with GoPie in the future.

Fourth, while our approach reduces the search space as mentioned in Section III-D, the search space can still be relatively large for some cases. To address this issue, DPOR [48] recognizes and explores only the representatives of equivalent interleavings, thus effectively reducing the number of interleavings that need to be explored. Incorporating the DPOR technique into our approach may yield better performance and reduce the computational resources required for testing. We are committed to exploring this possibility in the future and improving the efficiency of our approach.

VI. Related Work

A. Coverage-guide Testing

Coverage-guided testing is a popular software testing technique used to automatically generate and execute inputs to identify bugs, vulnerabilities, and other issues in a program. Tools such as *AFL* [49], *libFuzzer* [42], and *honggfuzz* [5] are commonly used for this purpose. However, these tools are limited when it comes to concurrency testing, primarily due to the gap between coverage-guided fuzzing and concurrency testing. Recent works [50], [51], [52], [53], [54] have improved the coverage-guide fuzzing approach by incorporating a concurrency-sensitive metric in place of the traditional coverage metric. However, these approaches rely on random scheduling techniques which can sometimes be ineffective, as highlighted in recent studies [7], [12], [55], [56], [57].

There are also some approaches that focus on effectively generating unit tests [43], [44], [45], [46]. Our approach differs from the aforementioned approaches in that they focus on generating effective concurrency test cases. Instead, we use a set of predetermined test cases and aim to enhance the concurrency coverage by optimizing the scheduling of concurrency execution orders. Therefore, our method is complementary to those approaches and can be used in conjunction with them to further improve the overall quality of concurrency testing.

B. Controlled Concurrency Testing

The field of data race detection has attracted extensive research efforts [12], [17], [51], [54], [55], [58]. GoPie is similar to previous techniques in terms of the aspects of thread scheduling or controlled concurrency testing [7], [12], [58], [55] as they all use the way to delay or schedule the

TABLE IX: The Performance of GoPie and the Variants. $Interleaving_1$ and $interleaving_2$ denotes the cross-routine and intra-routine interleaving. '-' means there is no successful scheduling chain generated. *Critical Channel State* includes those status such as *full*, *empty* and *closed* of channels.

Package	Interleaving$_1$			Interleaving$_2$			Valid Scheduling Chain			Critical Channel State		
Group	GoPie	GoPie-FB	GoPie-Mu	GoPie	GoPie-FB	GoPie-Mu	GoPie	GoPie-FB	GoPie-Mu	GoPie	GoPie-FB	GoPie-Mu
go-ethereum/console	7,792	7,586	6,080	13,380	9,488	8,543	10,982	1,345	-	168	145	141
go-ethereum/eth/fetcher	2,937	2,856	2,834	3,856	3,678	3,636	275	17	-	143	139	139
grpc-go/stats/opencensus	6,313	5,274	3,516	11,214	10,980	6,947	5,273	334	-	225	210	194
grpc-go/xds	650	274	277	796	505	507	7,024	-	-	36	36	36
etcd/server/etcdserver	7,720	2,650	3,309	9,850	6,383	7,193	39,718	81	-	259	258	259
etcd/tests/integration/snapshot	65,915	65,872	65,904	71,409	71,239	71,075	188	24	-	421	412	415
etcd/tests/integration/proxy/grpcproxy	37,747	37,745	37,795	33,521	33,080	33,273	321	29	-	403	382	380
k8s.io/client_go/tools/watch	2,792	1,441	1,375	2,911	2,159	2,140	6,678	5	-	116	116	116

threads to conduct interleaving exploration. However, these techniques mainly focus on memory access, and try to trigger bugs such as data races, which is not suitable for detecting concurrency primitive-related bugs like communication deadlock. For instance, *CONZZER* [7] is a tool designed specifically for detecting concurrency bugs in C/C++ programs. It utilizes context-sensitive *call pairs* and *directed scheduling* to explore potential concurrent function calls and identify possible concurrency issues. Our approach shares similar ideas with *CONZZER* in terms of directed scheduling. The key difference is our approach schedules on the operation level instead of the function level. And we incorporate certain constraints on Go concurrent primitives during mutation to skip unnecessary schedulings, which is more efficient. In Section III-D, we conduct a constraint to reduce the search space. Our approach shares the same goal with the technique of partial order reduction (POR) [20], [48]. However, while POR seeks to identify and skip equivalent interleavings, we instead aim to divide the events into separate sets to reduce the search space.

C. Go-specific Concurrency Bug Detection

Recent research on concurrency testing in Go has primarily concentrated on static analysis techniques [22], [23], [24], [59], [60], [61], [62], [63]. Despite their popularity, these static analysis techniques have certain limitations. For instance, they are prone to generating false positives and are not scalable for testing large programs. Even the latest state-of-the-art method Goat [23] has reported a high rate of failure, quitting on 70% of real-world project evaluations, and detecting more than 30% of false positives. Various dynamic bug detection tools are also being used by industries for Go-specific concurrency issues. For instance, Go comes with a built-in global deadlock detector that reports errors when all the goroutines are blocked [64]. Moreover, *go-deadlock* [65] is another deadlock detector that changes the package of locks to lock-sensitive ones. Additionally, many renowned open-source Golang projects have their own leak detectors, such as *goleak* [27] used by Uber and *leaktest* used by *cockroachdb* [66]. Despite this, these tools do not increase the probability of discovering new bugs through scheduling. Go has its own fuzzing framework [67], but like traditional fuzzing tools, it targets user input for mutation rather than concurrent interleaving, so it cannot be applied to concurrent testing scenarios.

Recently, some researchers have started to shift their focus towards using scheduling methods for exploring hard-to-find interleavings and uncovering Go-specific concurrency bugs. For instance, GoAT [11] uses random delay injection to schedule a concurrency program. However, recent studies [12], [13] have identified that such exploration is often inefficient as it frequently explores duplicate interleavings while missing the hard-to-find ones. GFuzz [14] tailors the idea of message reordering to directional control the execution orders of channel operations inside `select` statements. However, it lacks the ability to control other primitives such as *lock* and *WaitGroup*, as well as channel operations outside `select`. In contrast, GoPie improves upon these limitations through the utilization of more generalized scheduling and more effective interleaving exploring methods. Additionally, the coverage feedback, `ChOpPair` in GFuzz, is mainly designed as a metric to measure the scheduling effects while cannot be utilized to infer new potential interleavings.

VII. Conclusion

This paper presents a novel dynamic testing technique GoPie for detecting Go concurrency bugs via directional primitive-constraint interleaving exploration. In particular, GoPie proposes a general scheduling method for Go concurrency testing and an efficient interleaving exploration strategy by learning from previous executions instead of exploring exhaustively or via random delay injection. We have evaluated GoPie on the Go-specific benchmark GoBench and six large-scale open-source projects. Experimental results show that GoPie significantly outperforms the baselines in terms of the number of detected bugs on GoBench. Besides, it has discovered 11 previously unknown unique concurrency bugs that are missed by existing state-of-the-art tools.

Acknowledgement

We sincerely thank all anonymous reviewers for their valuable comments. This work was supported by the National Natural Science Foundation of China (Grant No. 62002125), the Young Elite Scientists Sponsorship Program by CAST (Grant No. 2021QNRC001) as well as the Hubei Province Key R&D Technology Special Innovation Project under Grant No.2021BAA032.

References

[1] T. Yuan, G. Li, J. Lu, C. Liu, L. Li, and J. Xue, "Gobench: A benchmark suite of real-world go concurrency bugs," in *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, J. W. Lee, M. L. Soffa, and A. Zaks, Eds. IEEE, 2021, pp. 187–199. [Online]. Available: https://doi.org/10.1109/CGO51591.2021.9370317

[2] N. Dilley and J. Lange, "An empirical study of messaging passing concurrency in go projects," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, X. Wang, D. Lo, and E. Shihab, Eds. IEEE, 2019, pp. 377–387. [Online]. Available: https://doi.org/10.1109/SANER.2019.8668036

[3] T. Tu, X. Liu, L. Song, and Y. Zhang, "Understanding real-world concurrency bugs in go," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds. ACM, 2019, pp. 865–878. [Online]. Available: https://doi.org/10.1145/3297858.3304069

[4] A. Fioraldi, D. C. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Y. Yarom and S. Zennou, Eds. USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[5] Google, "honggfuzz," https://honggfuzz.dev/, 2017.

[6] W. Luo and B. Demsky, "C11tester: a race detector for C/C++ atomics," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds. ACM, 2021, pp. 630–646. [Online]. Available: https://doi.org/10.1145/3445814.3446711

[7] Z. Jiang, J. Bai, K. Lu, and S. Hu, "Context-sensitive and directional concurrency fuzzing for data-race detection," in *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/auto-draft-198/

[8] M. Christiaens and K. D. Bosschere, "Trade: Data race detection for java," in *Computational Science - ICCS 2001, International Conference, San Francisco, CA, USA, May 28-30, 2001. Proceedings, Part II*, ser. Lecture Notes in Computer Science, V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. J. K. Tan, Eds., vol. 2074. Springer, 2001, pp. 761–770. [Online]. Available: https://doi.org/10.1007/3-540-45718-6_81

[9] Y. Cai, H. Yun, J. Wang, L. Qiao, and J. Palsberg, "Sound and efficient concurrency bug prediction," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 255–267. [Online]. Available: https://doi.org/10.1145/3468264.3468549

[10] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, "Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, T. Brecht and C. Williamson, Eds. ACM, 2019, pp. 162–180. [Online]. Available: https://doi.org/10.1145/3341301.3359638

[11] S. Taheri and G. Gopalakrishnan, "Goat: Automated concurrency analysis and debugging tool for go," in *IEEE International Symposium on Workload Characterization, IISWC 2021, Storrs, CT, USA, November 7-9, 2021*. IEEE, 2021, pp. 138–150. [Online]. Available: https://doi.org/10.1109/IISWC53511.2021.00023

[12] A. Choudhary, S. Lu, and M. Pradel, "Efficient detection of thread safety violations via coverage-guided generation of concurrent tests," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 266–277. [Online]. Available: https://doi.org/10.1109/ICSE.2017.32

[13] V. Terragni and M. Pezzè, "Effectiveness and challenges in generating concurrent tests for thread-safe classes," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 64–75. [Online]. Available: https://doi.org/10.1145/3238147.3238224

[14] Z. Liu, S. Xia, Y. Liang, L. Song, and H. Hu, "Who goes first? detecting go concurrency bugs via message reordering," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 888–902. [Online]. Available: https://doi.org/10.1145/3503222.3507753

[15] M. Emmi, S. Qadeer, and Z. Rakamaric, "Delay-bounded scheduling," in *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, T. Ball and M. Sagiv, Eds. ACM, 2011, pp. 411–422. [Online]. Available: https://doi.org/10.1145/1926385.1926432

[16] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," pp. 446–455, 2007. [Online]. Available: https://doi.org/10.1145/1250734.1250785

[17] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," pp. 167–178, 2010. [Online]. Available: https://doi.org/10.1145/1736020.1736040

[18] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel," in *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, R. H. Arpaci-Dusseau and B. Chen, Eds. USENIX Association, 2010, pp. 151–162. [Online]. Available: http://www.usenix.org/events/osdi10/tech/full_papers/Erickson.pdf

[19] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 267–280. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf

[20] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, ser. Lecture Notes in Computer Science. Springer, 1996, vol. 1032. [Online]. Available: https://doi.org/10.1007/3-540-60761-7

[21] Google, "Effective go: Concurrency," https://golang.org/doc/effective_go.html#concurrency, 2023.

[22] Z. Liu, S. Zhu, B. Qin, H. Chen, and L. Song, "Automatically detecting and fixing concurrency bugs in go software systems," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds. ACM, 2021, pp. 616–629. [Online]. Available: https://doi.org/10.1145/3445814.3446756

[23] O. H. Veileborg, G. Saioc, and A. Møller, "Detecting blocking errors in go programs using localized abstract interpretation," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 32:1–32:12. [Online]. Available: https://doi.org/10.1145/3551349.3561154

[24] N. Dilley and J. Lange, "Bounded verification of message-passing concurrency in go using promela and spin," Ph.D. dissertation, 2020. [Online]. Available: https://doi.org/10.4204/EPTCS.314.4

[25] K. Sen, "Effective random testing of concurrent programs," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds. ACM, 2007, pp. 323–332. [Online]. Available: https://doi.org/10.1145/1321631.1321679

[26] Google, "Data race detector," https://go.dev/doc/articles/race_detector, 2023.

[27] Uber, "goleak," https://github.com/uber-go/goleak, 2021.

[28] https://github.com/ethereum/go-ethereum/issues/27247.

[29] Google, "Package ast," https://golang.org/pkg/go/ast/, 2023.

[30] "grpc_6190," https://github.com/grpc/grpc-go/pull/6190.

[31] "grpc_6174," https://github.com/grpc/grpc-go/pull/6174.

[32] "grpc_6172," https://github.com/grpc/grpc-go/pull/6172.

[33] "etcd_15785," https://github.com/etcd-io/etcd/pull/15785.

[34] "etcd_15675," https://github.com/etcd-io/etcd/pull/15675.

[35] "etcd_15723," https://github.com/etcd-io/etcd/issues/15723.

[36] "k8s_117517," https://github.com/kubernetes/kubernetes/issues/117517.

[37] "k8s_117534," https://github.com/kubernetes/kubernetes/pull/117534.

[38] "k8s_114071," https://github.com/kubernetes/kubernetes/pull/114071.

[39] "eth_27675," https://github.com/ethereum/go-ethereum/pull/27695.

[40] "eth_27110," https://github.com/ethereum/go-ethereum/issues/27110.

[41] N. Dilley and J. Lange, "Bounded verification of message-passing concurrency in go using promela and spin," in *Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2020, Dublin, Ireland, 26th April 2020*, ser. EPTCS, S. Balzer and L. Padovani, Eds., vol. 314, 2020, pp. 34–45. [Online]. Available: https://doi.org/10.4204/EPTCS.314.4

[42] LLVM, "libFuzzer–a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, 2022.

[43] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Mseqgen: object-oriented unit-test generation via mining source code," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, H. van Vliet and V. Issarny, Eds. ACM, 2009, pp. 193–202. [Online]. Available: https://doi.org/10.1145/1595696.1595725

[44] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, P. Tonella and A. Orso, Eds. ACM, 2010, pp. 147–158. [Online]. Available: https://doi.org/10.1145/1831708.1831728

[45] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated unit test generation for classes with environment dependencies," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, I. Crnkovic, M. Chechik, and P. Grünbacher, Eds. ACM, 2014, pp. 79–90. [Online]. Available: https://doi.org/10.1145/2642937.2642986

[46] S. Wang, N. Shrestha, A. K. Subburaman, J. Wang, M. Wei, and N. Nagappan, "Automatic unit test generation for machine learning libraries: How far are we?" in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 1548–1560. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00138

[47] "Partial deadlock detector," https://github.com/golang/go/issues/13759.

[48] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," pp. 110–121, 2005. [Online]. Available: https://doi.org/10.1145/1040305.1040315

[49] M. Zalewski, "American fuzzy lop," 2017.

[50] C. Liu, D. Zou, P. Luo, B. B. Zhu, and H. Jin, "A heuristic framework to detect concurrency vulnerabilities," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 529–541. [Online]. Available: https://doi.org/10.1145/3274694.3274718

[51] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 754–768. [Online]. Available: https://doi.org/10.1109/SP.2019.00017

[52] N. Vinesh and M. Sethumadhavan, "Confuzz—a concurrency fuzzer," in *First International Conference on Sustainable Technologies for Computational Intelligence: Proceedings of ICTSCI 2019*. Springer, 2020, pp. 667–691.

[53] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1643–1660. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00078

[54] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu, "MUZZ: thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs," pp. 2325–2342, 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu

[55] Y. Cai and Q. Lu, "Dynamic testing for deadlocks via constraints," *IEEE Trans. Software Eng.*, vol. 42, no. 9, pp. 825–842, 2016. [Online]. Available: https://doi.org/10.1109/TSE.2016.2537335

[56] H. Jia, M. Wen, Z. Xie, X. Guo, R. Wu, M. Sun, K. Chen, and H. Jin, "Detecting JVM JIT compiler bugs via exploring two-dimensional input spaces," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 43–55. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00016

[57] M. Li, J. Cao, Y. Tian, T. O. Li, M. Wen, and S.-C. Cheung, "Comet: Coverage-guided model generation for deep learning library testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, jul 2023. [Online]. Available: https://doi.org/10.1145/3583566

[58] C. Wen, M. He, B. Wu, Z. Xu, and S. Qin, "Controlled concurrency testing via periodical scheduling," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 474–486. [Online]. Available: https://doi.org/10.1145/3510003.3510178

[59] N. Dilley and J. Lange, "Automated verification of go programs via bounded model checking," pp. 1016–1027, 2021. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678571

[60] D. Honnef, "Staticcheck – a collection of static analysis tools for working with go code," https://github.com/dominikh/go-tools, 2022.

[61] N. Ng and N. Yoshida, "Static deadlock detection for concurrent go by global session graph synthesis," in *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, A. Zaks and M. V. Hermenegildo, Eds. ACM, 2016, pp. 174–184. [Online]. Available: https://doi.org/10.1145/2892208.2892232

[62] N. Dilley and J. Lange, "Automated verification of go programs via bounded model checking," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 1016–1027. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678571

[63] J. Lange, N. Ng, B. Toninho, and N. Yoshida, "A static verification framework for message passing in go using behavioural types," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 1137–1148. [Online]. Available: https://doi.org/10.1145/3180155.3180157

[64] Google, "The go language," 2023, [Online; accessed 5 May 2023]. [Online]. Available: https://golang.org/

[65] Google, "Package deadlock," https://pkg.go.dev/github.com/sasha-s/go-deadlock, 2021.

[66] C. Labs, "leaktest," https://github.com/cockroachdb/cockroach/tree/master/pkg/util/leaktest, 2021.

[67] Google, "Go fuzzing," https://go.dev/security/fuzz/, 2023.